Práctica de diseño de un procesador monociclo

El objetivo de esta práctica de la asignatura de Tecnología de Computadores es la implementación de un procesador MIPS reducido (nombre del procesador **TG00MIPS**).

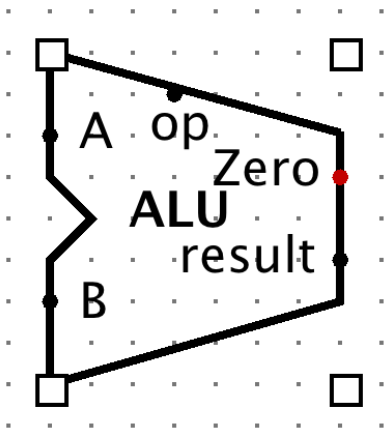
Las características hardware principales del procesador TG00MIPS son:

- Palabra de 32 bits.
- Banco de 32 registros de propósito general de 32 bits. **El registro $0 será de solo lectura y contendrá el valor 0**.
- Unidad aritmético-lógica (ALU) de 32 bits con las siguientes características:
  - Suma y resta de operandos enteros representados en convenio complemento a 2 (C2)
  - Operaciones lógicas AND, OR y NOR.
  - Operación de comparación "menor que".
  - Un flag de estado Z: indica (con un 1) que el resultado de la ALU es igual a 0.
- Memoria de instrucciones:
  - Bus de direcciones y de datos de 32 bits
  - Direccionable a nivel de byte.
- Memoria de datos:
  - Bus de direcciones y de datos de 32 bits
  - Direccionable a nivel de byte.

Para el diseño del procesador se proporcionan ya implementados los siguientes elementos: una ALU, un banco de 32 registros de 32 bits y las memorias de instrucciones y de datos.

En las siguientes figuras y tablas se muestra la forma, interfaz y funcionamiento de los elementos proporcionados.

ALU (ALU)

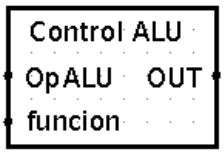


Señales	
A[31:0]	Dato de entrada
B[31:0]	Dato de entrada
op[3:0]	Operación a realizar por la ALU
result[31:0]	Dato de salida
Zero	Se activa (1) cuando el resultado es cero

op[3:0]	Operación ALU	
0000	result = A & B	And (bit a bit)
0001	result = A | B	Or (bit a bit)
0010	result = A + B	Suma
0110	result = A – B	Resta
0111	If (A<B) result = 1 else result = 0	Activar si A menor que B
1100	result = $\overline{A \mid B}$	Nor (bit a bit)

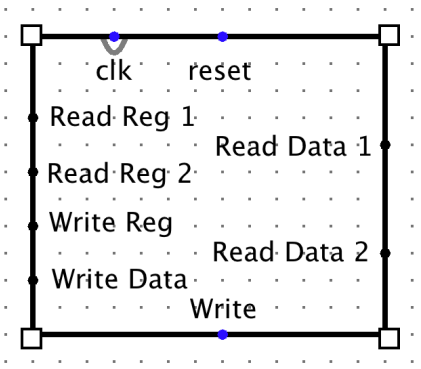
Control ALU

Señales	
OpALU[2:0]	Selección de operación de la ALU
funcion	Código de función de las instrucciones ALU
OUT[3:0]	Código de operación para la ALU (op)



OpALU[2:0]	funcion	OUT
000	xxxxxx	0010 (ALU suma)
001	xxxxxx	0110 (ALU resta)
010	xxxxxx	0000 (ALU and)
011	xxxxxx	0001 (ALU or)
100	xxxxxx	0111 (ALU activar si menor)
101	xxxxxx	1100 (ALU nor)
111 (funcion)	100000	0010 (ALU suma)
	100010	0110 (ALU resta)
	100100	0000 (ALU and)
	100101	0001 (ALU or)
	101010	0111 (ALU activar si menor)

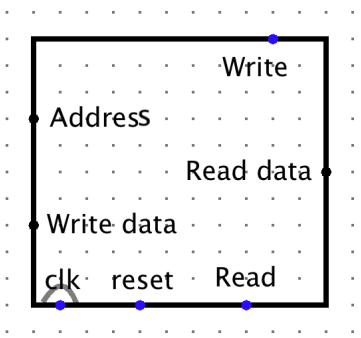
BANCO DE REGISTROS (Register file)



Señales	
clk	Señal de reloj para sincronizar la escritura
Write	Señal de escritura (1)
Read Data 1[31:0]	Dato leído del registro indicado en Read Reg 1
Read Data 2[31:0]	Dato leído del registro indicado en Read Reg 2
Write Data[31:0]	Dato a escribir en el registro indicado en Write Reg
Read Reg 1[5:0]	Código del registro a leer por salida Read Data 1
Read Reg 2[5:0]	Código del registro a leer por salida Read Data 2
Write Reg[5:0]	Código del registro a escribir

Write	Operación del registro
0	Lectura: Read Data 1[31:0] = Contenido almacenado en el registro indicado por Read Reg 1[5:0] Read Data 2[31:0] = Contenido almacenado en el registro indicado por Read Reg 2[5:0]
1	Escritura por flanco de subida de CLK: Registro indicado por Write Reg[5:0] se carga con Write Data[31:0] Además se realizan las dos lecturas al igual que con Write=0

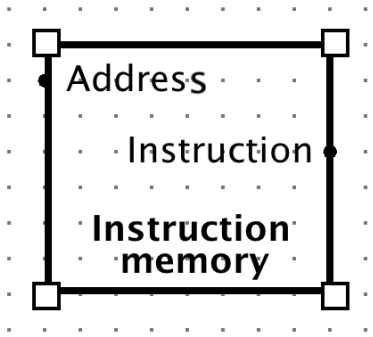
Memoria de datos (Data Memory 32)



Señales	
clk	Señal de reloj para sincronizar las escrituras en memoria
Write	Señal de escritura
Read	Señal de lectura
Read data[31:0]	Bus de datos de salida (lectura)
Write data[31:0]	Bus de datos de entrada (escritura)
Address[31:0]	Bus de direcciones

Write	Read	Operación de la memoria
0	1	Lectura: Read data[31:0] = M[Address[31:0]]
1	0	Escritura por flanco de subida de clk: M[Address[31:0]] ← Read data[31:0]

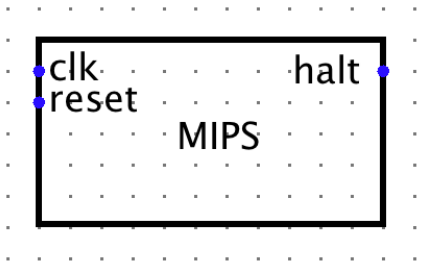
Memoria de Instrucciones (Instruction memory)



Señales	
Instruction[31:0]	Bus de datos de salida (lectura)
Address[31:0]	Bus de direcciones

Operación de la memoria
Continuamente: Instruction[31:0] = M[Address[31:0]]

El interfaz del procesador a diseñar es el siguiente:



Entradas:

- clk: reloj del sistema que sincronizará la ejecución de todas las instrucciones
- reset: puesta a cero de todos los elementos de memoria del procesador

Salida:

- halt: señal que se activará cuando se ejecute una instrucción especial denominada “halt”.

INSTRUCCIONES

El conjunto mínimo de instrucciones que debe soportar el procesador son:

- **Instrucciones de Bifurcación:**
  - Salto incondicional: **J dir**

Instr. J: J dir : PC ← (PC+4)[31:28],dir,00

  - Salto condicional: **BEQ rs, rt, dir**

Instr. BEQ: BEQ rs,rt,dir : if (R[rs]=R[rt]) PC ← (PC+4) + (ExSig(dir)<<2)

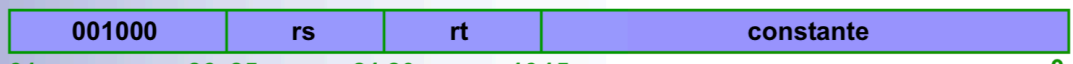
- **Instrucciones de transformación de información:**
  - Operaciones con la ALU: **OpAlu rd, rs, rt**

Instr. ALU: OpAlu rd,rs,rt : R[rd] ← R[rs] (funcion) R[rt]

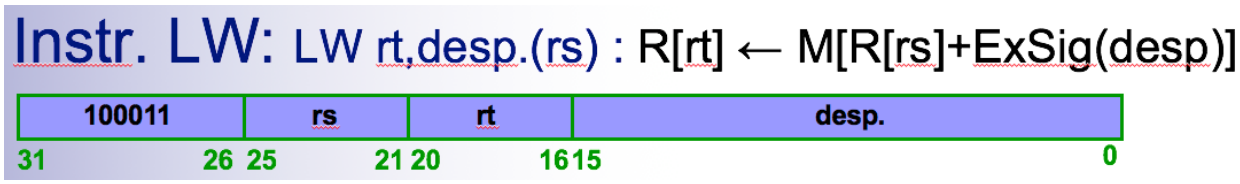
OpAlu	función	OpAlu	función
ADD	100000	OR	100101
SUB	100010	SLT	101010
AND	100100		

  - Suma con inmediato: **ADDI rt, rs, constante**

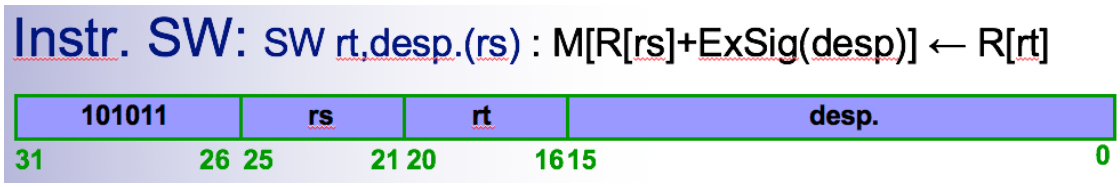
Instr. ADDI: ADDI rt, rs, constante : R[rt] ← R[rs] + ExSig(constante)



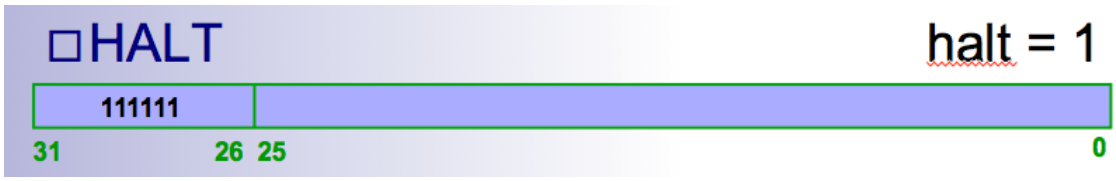
- **Instrucciones de transferencia:**
  - Carga: **LW rt, desp.(rs)**



  - Almacenamiento: **SW rt, desp.(rs)**



- **Instrucción especial:**
  - **HALT**



Tareas a desarrollar por el alumno:

1. Diseño e implementación de las instrucciones J y HALT

Empezarás la implementación del procesador consiguiendo que sea capaz de ejecutar estas instrucciones. Debes utilizar como bloques básicos de diseño: ALU, banco de registros, registros, sumador/restador, biestables, multiplexores, codificadores, decodificadores y puertas lógicas (todos estos elementos con el ancho de bits que necesites).

Debes bajarte de la plataforma de enseñanza virtual un prototipo de proyecto que irás modificando para diseñar tu procesador. El nombre del prototipo es TG00MIPS.ZIP. Dentro de dicho fichero comprimido encontrarás el fichero con el circuito base TG00MIPS.circ. Dicho nombre deberá ser cambiado antes de empezar a trabajar con él. Deberás cambiar los 4 primeros caracteres (**TG00**) por los que correspondan:

- **T**: Titulación del alumno (**I** Ing. Informática, **S** Ing. del Software, **C** Ing. de Computadores)
- **G**: Grupo (**A, B, …**)
- **00**: Número de equipo del alumno con dos dígitos (**01, 02, …**)

Así por ejemplo el equipo 13 del grupo B de la ingeniería informática, deberá trabajar sobre el proyecto con nombre: **IB13MIPS.circ.**

1.1. Implementación del hardware para el fetching de las instrucciones:

Debes implementar en tu circuito el hardware necesario para la lectura secuencial de las instrucciones de la memoria de instrucciones. Tienes que usar un registro que haga la función de contador de programa (PC): debe direccionar la memoria de instrucciones y actualizarse en cada flanco activo del reloj con la dirección de la siguiente instrucción (PC+4), añadiendo para ello el hardware adicional que sea necesario. **Recuerda que para que un registro cargue un valor necesita recibir un flanco por su señal de sincronismo (clk).**

Comprobación del funcionamiento: Para comprobar el funcionamiento del hardware de lectura de las instrucciones, debes introducir un programa en la memoria de instrucción y comprobar que se van leyendo en secuencia una a una las instrucciones almacenadas en la memoria, con cada flanco activo del reloj.

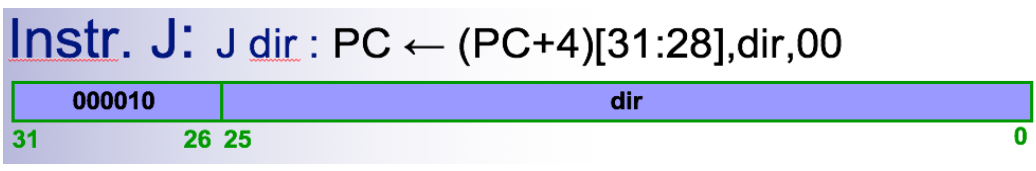
NOTA: La instrucción denominada NOP es una instrucción que no hace nada y se codifica con todos sus 32 bits a 0.

Introduce el siguiente código en la memoria de instrucciones (TG00ROM.txt):

Dir.(hex)	Instrucciones	Código binario (hex) (contenido TG00ROM.txt)
0	nop	00000000
4	nop	00000000
8	j cinco	08000005
C	tres: nop	00000000
10	j siete	08000007
14	cinco: nop	00000000
18	j tres	08000003
1C	siete: nop	00000000
20	halt	fc000000

Comprueba que se van leyendo las instrucciones en la misma secuencia que están almacenadas en memoria (aparecen a la salida de la memoria de instrucción). **El retardo de lectura de la memoria es de 4 ticks, asegúrate que el ciclo del reloj principal del procesador (en la hoja raíz) tenga más de 4 ticks entre dos flancos activos (la duración del nivel alto del reloj más el nivel bajo debe sumar al menos 5 ticks).**

1.2. Implementación de la instrucción J



Implementa el hardware necesario para que, una vez leída la instrucción de memoria, si es una instrucción J (opcode 000010) cargue en PC, en vez de PC+4, la dirección de salto de 32 bits construida concatenando los cuatro bits más significativos de PC+4, seguidos de los 26 bits del campo de operando *dir*, seguidos a su vez de dos ceros.

Recuerda que, si la instrucción no es una J, PC se debe cargar con PC+4 como hacía en el apartado anterior.

En la unidad de datos deberás crear el camino HW que permita construir la dirección de destino y llevarla a la entrada de datos de PC sin que se deje de poder cargar en PC lo que se le había conectado previamente.

En la unidad de control deberás reconocer (decodificar) que es una instrucción J y activar el/los punto/s de control que activan el camino de datos HW que lleva la dirección de salto a PC (en vez de PC+4).

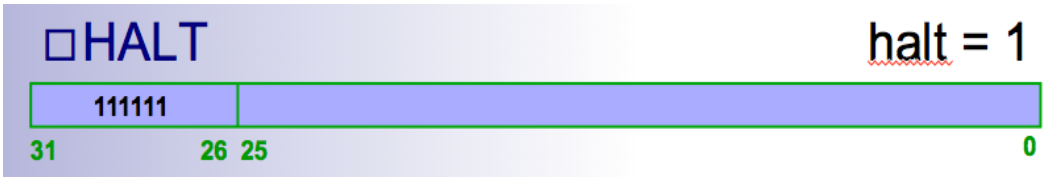
Comprobación del funcionamiento: Para comprobar el funcionamiento de la instrucción J introduce el mismo programa anterior en la memoria de instrucciones:

Dir.(hex)	Instrucciones	Código binario (hex) (contenido TG00ROM.txt)
0	nop	00000000
4	nop	00000000
8	j cinco	08000005
C	tres: nop	00000000
10	j siete	08000007
14	cinco: nop	00000000
18	j tres	08000003
1C	siete: nop	00000000
20	halt	fc000000

Generando ticks de reloj, debes comprobar que las instrucciones J se están ejecutando (ahora PC no avanzará secuencialmente y por tanto las instrucciones no se leerán en orden secuencial). La traza (traza = secuencia de valores que van tomando los elementos) para PC y la salida de la memoria de instrucción debería verse así:

PC	Salida mem instrucción
0	00000000
4	00000000
8	08000005
14	00000000
18	08000003
C	00000000
10	08000007
1C	00000000
20	fc000000

1.3. Implementación de la instrucción HALT



Observando la traza del apartado anterior te habrás dado cuenta de que tras la instrucción HALT de la posición $20_{(16}$ de memoria, el procesador ha seguido leyendo instrucciones (ceros) secuencialmente. Con la implementación de la instrucción HALT podremos parar el procesador. Para ello, aunque se sigan produciendo ticks, no deben llegar flancos de reloj al procesador. Esto se consigue activando la señal de salida "halt" del circuito cuando se detecta que la instrucción que tiene que ejecutarse es un HALT (opcode 111111).

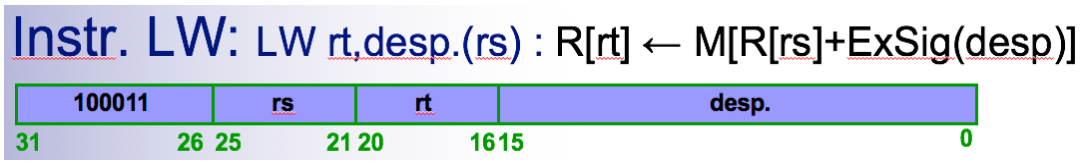
En la unidad de control decodifica esta instrucción, activando el punto de control "halt" como resultado.

Comprobación del funcionamiento: Simula el mismo programa anterior y comprueba como ahora tras la instrucción HALT (PC = $20_{(16)}$) ya no se ejecutan más instrucciones (PC se queda con el valor $24_{(16}$ por muchos ticks que se produzcan).

2. Implementación de las instrucciones LW y SW

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir las instrucciones LW y SW. Recuerda que debes añadir el nuevo HW y conexiones que consideres necesarios en tu procesador, pero manteniendo la posibilidad de ejecutar las instrucciones que has implementado previamente en el mismo (J y HALT).

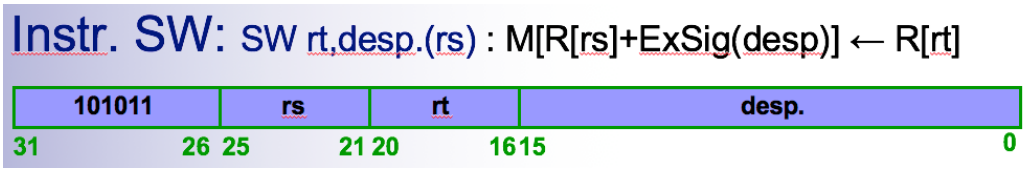
2.1. Implementación de la instrucción LW



Esta instrucción lee de la memoria de datos un valor para escribirlo en el registro destino indicado por *rt*. La dirección de acceso a memoria se calcula sumando en la ALU el contenido del registro especificado por el campo de operando *rs* con el resultado de realizar la extensión de signo a 32 bits del campo de operando *desp*,

Recuerda que el banco de registros y la memoria de datos necesitan una señal de reloj (clk) para funcionar.

2.2. Implementación de la instrucción SW



Esta instrucción lee el contenido del registro indicado por el operando fuente *rt* para escribirlos en la memoria de datos. La dirección de acceso a memoria se calcula en la posición calculada sumando en la ALU el contenido del registro especificado por el campo de operando *rs* con el resultado de realizar la extensión de signo a 32 bits del campo de operando *desp*.

Recuerda que el banco de registros y la memoria de datos necesitan una señal de reloj (clk) para funcionar.

2.3. Comprobación del funcionamiento de las instrucciones LW y SW

Introduce el siguiente programa en la memoria de instrucciones (TG00ROM.txt) y los siguientes datos en la memoria de datos (TG00RAM.txt).

Memoria de instrucciones

Dir.(hex)	Instrucciones	Contenido TG00ROM.txt
0	lw $1, 0($0)	8c010000
4	lw $2, 0($1)	8c220000
8	sw $1, 4($0)	ac010004
C	sw $2, 0($0)	ac020000
10	lw $2, 8($1)	8c220008
14	lw $1, 8($0)	8c010008
18	sw $2, 8($0)	ac020008
1C	sw $1, 0xC($0)	ac01000c
20	halt	fc000000

Memoria de datos

Dir.(hex)	Contenido TG00RAM.txt
0	00000004
4	00000003
8	00000002
C	00000001
10	00000000
14	
18	
1C	
20	

Simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, $1, $2 y las 4 primeras posiciones de memoria de datos (todos en hexadecimal).

Antes de poder simular correctamente, tienes que calcular el valor correcto de ticks que asignarle al reloj: debe darle tiempo a ejecutarse a la instrucción más lenta de las que has implementado. Comprueba el retardo sufrido en el camino que activa cada instrucción para su ejecución. Los retardos (en ticks) para las unidades funcionales que se te han proporcionado en el prototipo son los siguientes (considera 0 el retraso del resto de elementos que hayas añadido tú en el diseño):

- **Memoria de instrucciones: 4** (desde que se pone una dirección válida en la entrada de direcciones, hasta que aparece correctamente el dato almacenado en dicha dirección, hay que esperar 4 ticks)

- **Memoria de datos: 4** (tanto en lectura como en escritura. Para lectura hay que esperar 4 ticks desde que se pone la dirección hasta que está disponible el dato. En escritura hay que esperar 4 ticks desde que se pone la dirección y el dato a escribir, hasta que se ha llevado a cabo la escritura correctamente)
- **ALU: 4** (desde que se cambia alguna de sus entradas hasta que aparece el resultado correcto en la salida)
- **Banco de registros: 2** (tanto para lectura como para escritura)

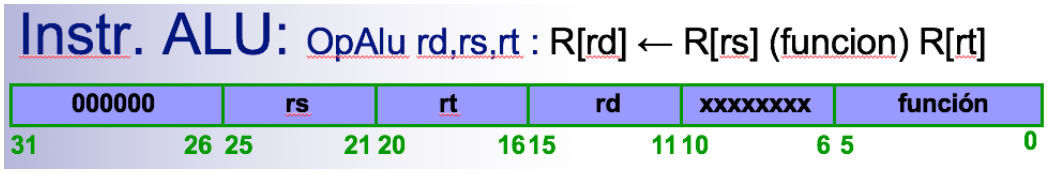
Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender cómo se ejecuta cada instrucción. Comprueba que lo que tú imaginas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila correspondiente al ciclo de instrucción/máquina de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

	PC	$1	$2	RAM[0]	RAM[1]	RAM[2]	RAM[3]
`lw $1, 0($0)`	00000000	00000000	00000000	00000004	00000003	00000002	00000001
`lw $2, 0($1)`	00000004	**00000004**	00000000	00000004	00000003	00000002	00000001
`sw $1, 4($0)`	00000008	00000004	**00000003**	00000004	00000003	00000002	00000001
`sw $2, 0($0)`	0000000c	00000004	00000003	00000004	**00000004**	00000002	00000001
`lw $2, 8($1)`	00000010	00000004	00000003	**00000003**	00000004	00000002	00000001
`lw $1, 8($0)`	00000014	00000004	**00000001**	00000003	00000004	00000002	00000001
`sw $2, 8($0)`	00000018	**00000002**	00000001	00000003	00000004	00000002	00000001
`sw $1, 0xC($0)`	0000001c	00000002	00000001	00000003	00000004	**00000001**	00000001
`halt`	00000020	00000002	00000001	00000003	00000004	00000001	**00000002**

3. Implementación de las instrucciones ALU y ADDI

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir las instrucciones ALU y ADDI. Recuerda que tus cambios no deben afectar a la correcta ejecución de las instrucciones que implementaste previamente (J, HALT, LW y SW).

3.1. Implementación de la instrucción ALU

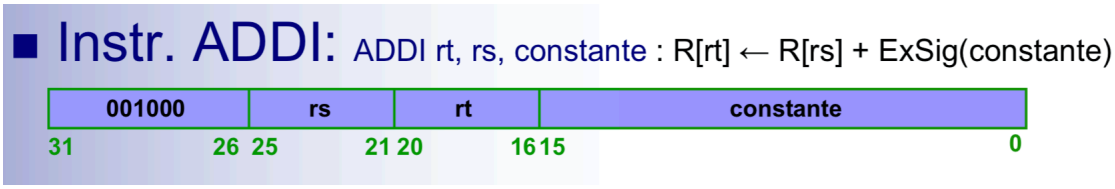


OpAlu	función	OpAlu	función
ADD	100000	OR	100101
SUB	100010	SLT	101010
AND	100100		

Esta instrucción debe llevar al registro indicado por el campo de operando *rd*, el resultado de operar en la ALU el contenido de los registros indicados por los campos de operando *rs* y *rt*. La operación que

debe realizar la ALU se especifica en el campo *función* de la instrucción máquina, según la tabla anterior.

3.2. Implementación de la instrucción ADDI



Esta instrucción lleva al registro especificado por el operando *rt*, el resultado de sumar en la ALU el contenido del registro especificado por el operando *rs* con el dato inmediato especificado en el campo de operando *constante*, extendido en signo hasta los 32 bits.

3.3. Comprobación del funcionamiento de las instrucciones ALU y ADDI

Introduce el siguiente programa en la memoria de instrucciones (TG00ROM.txt) y los siguientes datos en la memoria de datos (TG00RAM.txt).

Memoria de instrucciones

Dir.(hex)	Instrucciones	Contenido TG00ROM.txt
0	lw $1, 0($0)	8c010000
4	lw $2, 4($0)	8c020004
8	add $3, $1, $2	00221820
C	sub $3, $2, $1	00411822
10	addi $3, $1, -2	2023fffe
14	addi $3, $3, 3	20630003
18	addi $2, $0, 11	2002000b
1C	and $1, $3, $2	00620824
20	or $1, $3, $2	00620825
24	slt $1, $2, $3	0043082a
28	slt $1, $3, $2	0062082a
2C	sw $3, 4($0)	ac030004
30	halt	fc000000

Memoria de datos

Dir.(hex)	Contenido TG00RAM.txt
0	4
4	-1
8	0

Simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, $1, $2, $3 y las 2 primeras posiciones de memoria de datos (todos en hexadecimal).

Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender como se ejecuta cada instrucción. Comprueba que lo que tú piensas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

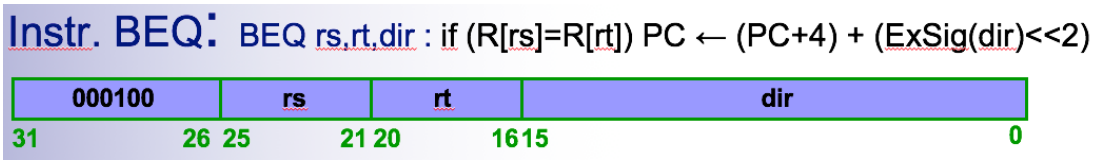
A la hora de analizar el resultado de las operaciones aritméticas (ADD, SUB, ADDI), ten en cuenta que la representación de los números enteros en el procesador utiliza el convenio C2 y que la visualización en la tabla está en hexadecimal (así por ejemplo el número -1, sería una cadena binaria de 32 1's, que representado en hexadecimal sería ffffffff)

	PC	$1	$2	$3	RAM[0]	RAM[1]
lw $1, 0($0)	00000000	00000000	00000000	00000000	00000004	ffffffff
lw $2, 4($0)	00000004	**00000004**	00000000	00000000	00000004	ffffffff
add $3, $1, $2	00000008	00000004	**ffffffff**	00000000	00000004	ffffffff
sub $3, $2, $1	0000000c	00000004	ffffffff	**00000003**	00000004	ffffffff
addi $3, $1, -2	00000010	00000004	ffffffff	**fffffffb**	00000004	ffffffff
addi $3, $3, 3	00000014	00000004	ffffffff	**00000002**	00000004	ffffffff
addi $2, $0, 11	00000018	00000004	ffffffff	**00000005**	00000004	ffffffff
and $1, $3, $2	0000001c	00000004	**0000000b**	00000005	00000004	ffffffff
or $1, $3, $2	00000020	**00000001**	0000000b	00000005	00000004	ffffffff
slt $1, $2, $3	00000024	**0000000f**	0000000b	00000005	00000004	ffffffff
slt $1, $3, $2	00000028	**00000000**	0000000b	00000005	00000004	ffffffff
sw $3, 4($0)	0000002c	**00000001**	0000000b	00000005	00000004	ffffffff
halt	00000030	00000001	0000000b	00000005	00000004	**00000005**

4. Implementación de la instrucción BEQ

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir la instrucción de salto condicional BEQ. Recuerda que deben seguirse ejecutando correctamente las instrucciones que implementaste previamente en el mismo (J, HALT, LW, SW, ALU y ADDI).

4.1. Implementación de la instrucción BEQ



Esta instrucción comprobará si el contenido de los registros especificados por los campos de operando rs y rt es el mismo. Si no es igual, la instrucción no hará nada (simplemente se incrementará PC: PC = PC + 4). En el caso de que el contenido sea igual, se ejecutará el salto, es decir, se cargará un valor en PC distinto de PC+4. El valor a cargar se calcula sumándole a (PC+4) el valor del campo de operando *dir* extendido en signo hasta 32 bits y desplazado dos bits a la izquierda (es decir, añadiéndole dos 0's en la posición menos significativa).

4.2. Comprobación del funcionamiento de la instrucción BEQ

Para comprobar el funcionamiento de esta instrucción utilizaremos un código que va a utilizar todas las instrucciones del procesador que hemos ido implementando, de manera que podamos comprobar que no

ha dejado de funcionar ninguna de ellas con las modificaciones que se han ido realizando.

Introduce el siguiente programa en la memoria de instrucciones (TG00ROM.txt) y los siguientes datos en la memoria de datos (TG00RAM.txt). El programa consta de un bucle que recorre secuencialmente un vector leyendo los valores almacenados para irlos sumando. Al terminar el recorrido del bucle, el resultado de la acumulación se almacena en memoria. Analiza el código, asegurándote de entender bien la función de cada uno de los registros (contador de elementos, acumulador de la suma, dirección del vector, ...) y qué hace exactamente cada una de las instrucciones para conseguir el valor de la suma total.

Memoria de instrucciones

Dir. (hex)	Instrucciones	Contenido TG00ROM.txt
0	lw $1, 0x14($0)	8c010014
4	addi $2, $0, 4	20020004
8	sub $3, $3, $3	00631822
C	add $4, $0, $0	00002020
10	lo: beq $4, $1, sl	10810005
14	lw $5, 0($2)	8c450000
18	add $3, $3, $5	00651820
1C	addi $2, $2, 4	20420004
20	addi $4, $4, 1	20840001
24	j lo	08000004
28	sl: sw $3, 0($0)	ac030000
2C	halt	fc000000

Memoria de datos

Dir. (hex)	Contenido TG00RAM.txt	
0	00000000	Result
4	fffffffe	Vector
8	00000004	
C	fffffffa	
10	00000008	
14	00000004	Size

Simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, $1, $2, $3, $4, $5 y la primera posición de memoria de datos (todos en hexadecimal).

Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender cómo se ejecuta cada instrucción. Comprueba que lo que tú piensas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

Ten en cuenta de nuevo, a la hora de interpretar la traza, que los datos del vector son números enteros de 32 bits representados en C2 y visualizados en hexadecimal.

		PC	$1	$2	$3	$4	$5	RAM[0]
	lw $1, 0x14($0)	00000000	00000000	00000000	00000000	00000000	00000000	00000000
	addi $2, $0, 4	00000004	**00000004**	00000000	00000000	00000000	00000000	00000000
	sub $3, $3, $3	00000008	00000004	**00000004**	00000000	00000000	00000000	00000000
	add $4, $0, $0	0000000c	00000004	00000004	**00000000**	00000000	00000000	00000000
lo:	beq $4, $1, sl	00000010	00000004	00000004	00000000	**00000000**	00000000	00000000
	lw $5, 0($2)	00000014	00000004	00000004	00000000	00000000	00000000	00000000
	add $3, $3, $5	00000018	00000004	00000004	00000000	00000000	**fffffffe**	00000000
	addi $2, $2, 4	0000001c	00000004	00000004	**fffffffe**	00000000	fffffffe	00000000
	addi $4, $4, 1	00000020	00000004	**00000008**	fffffffe	00000000	fffffffe	00000000
	j lo	00000024	00000004	00000008	fffffffe	**00000001**	fffffffe	00000000
lo:	beq $4, $1, sl	**00000010**	00000004	00000008	fffffffe	00000001	fffffffe	00000000
	lw $5, 0($2)	00000014	00000004	00000008	fffffffe	00000001	fffffffe	00000000
	add $3, $3, $5	00000018	00000004	00000008	fffffffe	00000001	**00000004**	00000000
	addi $2, $2, 4	0000001c	00000004	00000008	**00000002**	00000001	00000004	00000000
	addi $4, $4, 1	00000020	00000004	**0000000c**	00000002	00000001	00000004	00000000
	j lo	00000024	00000004	0000000c	00000002	**00000002**	00000004	00000000
lo:	beq $4, $1, sl	**00000010**	00000004	0000000c	00000002	00000002	00000004	00000000
	lw $5, 0($2)	00000014	00000004	0000000c	00000002	00000002	00000004	00000000
	add $3, $3, $5	00000018	00000004	0000000c	00000002	00000002	**fffffffa**	00000000
	addi $2, $2, 4	0000001c	00000004	0000000c	**fffffffc**	00000002	fffffffa	00000000
	addi $4, $4, 1	00000020	00000004	**00000010**	fffffffc	00000002	fffffffa	00000000
	j lo	00000024	00000004	00000010	fffffffc	**00000003**	fffffffa	00000000
lo:	beq $4, $1, sl	**00000010**	00000004	00000010	fffffffc	00000003	fffffffa	00000000
	lw $5, 0($2)	00000014	00000004	00000010	fffffffc	00000003	fffffffa	00000000
	add $3, $3, $5	00000018	00000004	00000010	fffffffc	00000003	**00000008**	00000000
	addi $2, $2, 4	0000001c	00000004	00000010	**00000004**	00000003	00000008	00000000
	addi $4, $4, 1	00000020	00000004	**00000014**	00000004	00000003	00000008	00000000
	j lo	00000024	00000004	00000014	00000004	**00000004**	00000008	00000000
lo:	beq $4, $1, sl	**00000010**	00000004	00000014	00000004	00000004	00000008	00000000
sl:	sw $3, 0($0)	**00000028**	00000004	00000014	00000004	00000004	00000008	00000000
	halt	0000002c	00000004	00000014	00000004	00000004	00000008	**00000004**

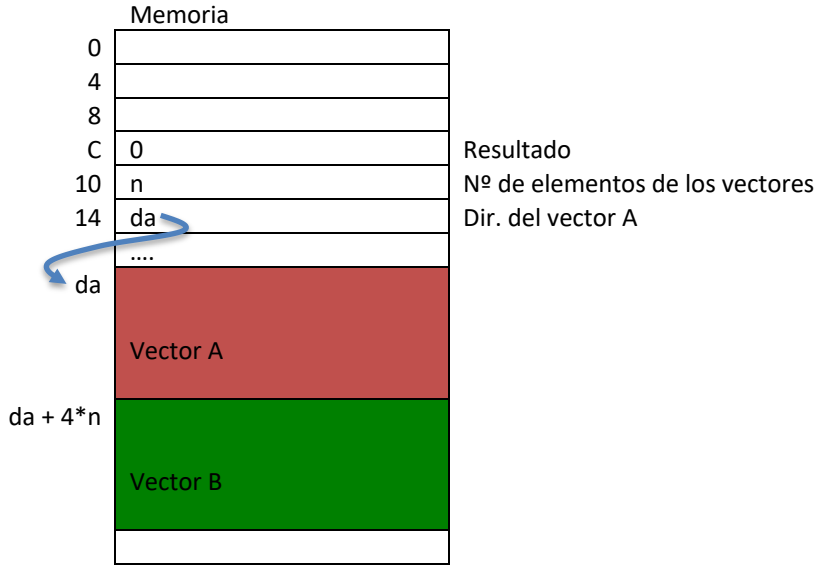
Práctica de programación del procesador

El objetivo de esta práctica es realizar un programa que se ejecute en el procesador diseñado.

El programa a realizar debe calcular la suma de las diferencias en valor absoluto de los elementos de dos vectores (SAD), operación que se utiliza para medir cuanto se parecen dos vectores A y B.

$$SAD(A,B) = \sum_i |A[i] - B[i]|$$

Estos dos vectores estarán formados cada uno por un número (n) de valores enteros (32 bits en C2) especificados en la posición $10_{(16}$ y almacenados consecutivamente a partir de la posición de memoria indicada en la dirección $14_{(16}$. Se realizará la resta elemento a elemento de los vectores (el primero de A menos el primero de B, el segundo menos el segundo, …) acumulando la suma total del valor absoluto de dichas restas. Debe almacenar el resultado en la posición de memoria $C_{(16}$ (no hay que tener en cuenta si se produce desbordamiento al calcular el SAD). Las posiciones de memoria $0_{(16}$, $4_{(16}$, y $8_{(16}$ se pueden usar para variables que necesite el programa. El esquema de la memoria de datos para este programa sería:



Diseña el programa propuesto utilizando las instrucciones disponibles en la implementación monociclo del procesador. Introdúcelo a partir de la posición 0 de la memoria de instrucciones (fichero TG00rom.txt).

Introduce los siguientes valores decimales (enteros de 32 bits) a partir de la posición $C_{(16}$ de la memoria de datos (fichero TG00ram.txt): 0, 4, 24, 25, -3, -6, 1, 23, -1, -4,-1. Esto corresponde a un ejemplo de vectores de 4 elementos.